1997

InterBase (KB): A Knowledge-based Multidatabase System for Data Warehousing

Nick Bassiliades

Ioannis Vlahavas

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Elias N. Houstis
Purdue University, enh@cs.purdue.edu

Report Number:
97-047

INTERBASE^{KB}: A KNOWLEDGE-BASED MULTIDATABASE SYSTEM FOR DATA WAREHOUSING

Nick Bassiliades
Ioannis Vlahavas
Ahmed K. Elmagarmid
Elias N. Houstis

CSD-TR #97-047
October 1997

InterBase^{KB}: A Knowledge-based Multidatabase System for Data Warehousing

[†]Nick Bassiliades[*], [†]Ioannis Vlahavas[**], [‡]Ahmed K. Elmagarmid[***] and [‡]Elias N. Houstis

[†]Dept. of Informatics
Aristotle University of Thessaloniki,
54006 Thessaloniki, Greece.
E-mail: {nbassili,vlahavas}@csd.auth.gr

[‡]Dept. of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398, USA
E-mail: {ake,enh}@cs.purdue.edu

[*] Supported by a scholarship from the Greek Foundation of State Scholarships (F.S.S. - I.K.Y.).

[**] This work was carried out while the author was on sabbatical leave at Purdue University.

[***] Partially supported by a grant from the Intel Corporation.

Abstract

This paper describes the extension of a multidatabase system with a knowledge-base module in order to support data warehousing. The multidatabase system integrates various component databases with a common query and transaction specification language, however, it does not provide capability for schema integration. The knowledge base module provides a declarative logic language with second-order syntax but first-order semantics for integrating the schemes of the data sources into the warehouse and for defining complex materialized views. Furthermore, the views are self-maintainable, i.e. they are incrementally maintained by an efficient mechanism that uses only the changes to the data sources.

Keywords: Multidatabase, Schema Integration, Data Warehouse, Materialized View, Knowledge Base System, Deductive Rule, and Active Rule

1. Introduction

A *Data Warehouse* is a repository that integrates information from data sources which may or may not be heterogeneous and makes them available for decision support querying and analysis [37]. There are two main advantages to data warehouses. First, they off-load decision support applications from the original, possibly on-line transaction, database systems. Second, they bring together information from multiple sources, thus providing a consistent database source for decision support queries.

Data warehouses store *materialized views* in order to provide fast and uniform access to information that is integrated from several distributed data sources. The warehouse provides a different way of looking at the data than the databases being integrated. Materialized views collect data from databases into the warehouse, but without copying each database into the warehouse. Queries on the warehouse can then be answered using the views instead of accessing the remote databases. When modification of data occurs on remote databases, they are transmitted to the warehouse. Incremental view maintenance techniques are used to maintain the views consistent with the modifications.

Multidatabase systems are confederations of pre-existing, autonomous and possibly heterogeneous database systems [31]. The pre-existing database systems that participate in the multidatabase are called *local* or *component database systems*. Usually the term "multidatabase" denotes *nonfederated* systems that integrate various heterogeneous database systems by supplying a common query language for specifying queries and transactions, without a global, integrated schema. On the other hand, *federated* database systems support partial or total integration of the schemata of their component database systems. However, federated databases do not materialize views but instead rely entirely upon the component database systems.

From the above definitions, it is evident that there are many similarities between multidatabases and data warehouses, and the former can therefore become the infrastructure for building and maintaining the latter. However, before a multi-database can be used for such a task, it must be supplied with powerful materialized view definition and maintenance tools in order to:

- keep the data stored at the warehouse consistent with the modifications of the data of the component databases, and
- provide to the user uniform and transparent access to the data of the underlying component databases.

In this paper we describe the extension of a nonfederated multidatabase system [28] with a knowledge-base module (KBM), thus providing a system with the necessary functionality to be used for data warehousing. The described system offers only the infrastructure for data warehousing and cannot be considered as a full warehouse system. The latter requires several back-end and front-end tools to ease the task of designing, administering, maintaining, and querying the data warehouse.

At the core of the KBM lies an active OODB with metaclasses [6] that supports events and event-driven rules and also integrates declarative rules (deductive and production rules). These features provide:

- languages and mechanisms for defining and self-maintaining complex materialized views, respectively, as well as
- rich structural and behavioral representation capabilities due to the powerful mechanism of metaclasses.

All these, combined with the interoperability offered at the application level and the high-level support for an integrated atomic commitment by the multidatabase system [29], make for a powerful yet flexible environment for data warehousing. Furthermore, the knowledge base module offers a flexible mechanism for defining and maintaining a global integrated schema for the multidatabase system.

Users and applications of the original multidatabase system can still work as they used to, but now can also access the materialized views inside the data warehouse. On the other hand, users and applications specifically written for the KBM can use extended features, such as declarative rules for expert database applications and semantic integrity enforcement, plus a pure OO view definition and maintenance mechanism.

The outline of this paper is as follows: Section 2 presents related work concerning multidatabase systems, schema integration, view definition and maintenance in OODBs; Section 3 overviews the architecture of the system; Section 4 describes the derived rule mechanism for defining and maintaining complex views; and Section 5 presents the method for translating and integrating the schemata of the component database systems. Finally Section 6 concludes this paper and discusses future work.

2. Related Work

This section briefly surveys several research issues and technologies that have contributed to the development of the InterBase^{KB} framework for data warehousing, namely a) object-oriented views, b) views for integrating heterogeneous data, and c) materialization of views. Furthermore, the relationships of these investigations with InterBase^{KB} are indicated. Finally, we briefly discuss the relationship of InterBase^{KB} with our previous research.

2.1 Object-Oriented Views

There are several research approaches to object-oriented views [1, 22, 33, 19]. The most important issue in object-oriented views is the preservation or non-preservation of the object identifiers (OIDs) of the base objects that are combined to produce the objects of the view. The *object-preserving* approach returns OIDs of base objects when the view is queried [33, 1] while the *object-generating* approach generates new OIDs to populate the view [1, 19]. The latter approaches also have techniques that allow for the "preservation" of the OIDs of the base objects. The OIDs of the view objects are represented as functions of the OIDs of the base objects. Therefore, by applying the inverse function, the OIDs of the base objects can be "calculated" from the OID of the view object.

Finally, the approach of [22] temporarily generates objects but not OIDs and deletes them after the query is answered. This approach also proposes to "store" the association between the OIDs of the base objects and the OIDs of the equivalent view objects. However, since the view objects are not assigned OIDs, this association is not implemented. Furthermore, despite that views can be defined using other views, it is not possible to store the association between objects of the two views even if the views were assigned OIDs. This is because view objects are volatile, and their OIDs differ every time they are temporarily created.

All of the above approaches (with the exceptions mentioned for [22]) allow direct updates to views that are propagated to base objects since the OIDs of the base objects are preserved in one way or another. Not all updates are allowed, however, as explained in Section 4.5.

Our approach is similar to the object-generating approaches except that we keep explicit references to the base object OIDs inside the `derivator` attribute of the view objects instead using OID generating functions. This is simpler and more practical to implement. Furthermore, it does feature the main advantage of the object-generating approach, namely the fast retrieval queries to the view objects, while it also keeps the basic advantage of the object-preserving approach, the updateability of the view objects. A notable difference with all of the above approaches is that they use a relational language for defining the view while we use a more complex first-order language that allows for definition of recursive views.

Concerning the placement of the view classes in the hierarchy lattice of the OODB schema, the approach of [19] does not allow the "appearance" of the original classes and the view classes together in the same application; therefore, the type hierarchy that the view introduces does not interfere with the OODB schema of base classes. The approach of [22] separates the view class hierarchy from the base class hierarchy, and the burden of ensuring the correctness of the view hierarchy falls upon the user.

The object-preserving approach of [1] incorporates the view class inside the existing class hierarchy using the specialization and generalization constructors to infer the place of insertion. Furthermore, it resolves the problem of duplicate objects (see Section 4.3) by making a clear distinction between real and virtual objects. However, the object-generating technique of imaginary objects that is also presented in [1] does not place the imaginary classes in the class hierarchy at all. Instead, they are placed in the schema as distinct, isolated classes. Finally, a clear distinction between types and classes in [33] allows for the successful placement of a view class in both hierarchies.

Our approach is different from all of the above for two reasons: a) our OODB does not support separation of types and classes and b) the materialized view classes are placed into an existing class hierarchy. In order to "emulate" type hierarchies, we use abstract classes, i.e. classes that do not have direct instances but rather serve as attribute and method holders. Furthermore, the extents of the base and view classes are not directly related in order to avoid duplicate objects at the same class hierarchy.

2.2 Schema Integration

There are two broad categories for integrating the schemata of individual, heterogeneous databases.

Approaches based on defining a common data model. Most of the existing approaches for schema integration belong to this category [31, 21, 20, 3]. The databases participating in the federation are mapped to a common data model (most commonly an object-oriented one) which acts as an "interpreter" among them. Furthermore, a common view that hides the structural differences on the schemas of the heterogeneous databases offers integration transparency. The major problem associated with the approaches in this category is the amount of human participation required for obtaining the mappings between the schemas of the common data model and the data models of the heterogeneous databases.

Approaches based on higher-order logics. The second approach for schema integration involves defining a higher-order language that can express relationships between the meta-information corresponding to the schemata of the individual databases [23, 25]. Thus, a common data model is not required, but the higher-order language plays the role of the data model. The major advantage of this approach is the declarativity it derives from its logical foundation. The approaches

above, however, are targeted towards the integration of heterogeneous relational databases into a multidatabase using views with object-preserving semantics [26].

Finally, an interesting approach that falls in between the above two categories is [4], an extensible logic-based meta-model that is able to capture the syntax and semantics of various data models. However, the second-order nature of this approach probably makes impractical an efficient implementation on a real database system; in addition the semantics are quite complex.

Our approach, a combination of the above approaches, is targeted towards the integration of heterogeneous data sources with varying data models through materialized views in a data warehouse. More specifically, we provide a declarative logic-based language with second-order semantics which is translated (using the metaclass schema of the OODB) into a set of first-order deductive rules. These deductive rules define a common view which is the global schema for the integrated heterogeneous databases using the default view definition and incemental maintenance mechanism of the system. In this way, we combine the declarativeness of logic and the flexibility of second-order data models and syntax, along with the rich structural and behavioral capabilities of an object-oriented data model. Finally, we efficiently implement the schema integration mechanism using the event-driven mechanisms of an active database.

Another interesting use of logic in multidatabase systems is querying and transaction specification. The concurrent logic programming language VPL [24] provides the specification of compensating actions of committed clauses and, therefore, is a declarative vehicle upon which advanced transaction models for multidatabase systems, such as Flex [14], can be both seamlessly defined and efficiently executed. Furthermore, dynamic, parallel querying and static, sequential schema integration can be also easily expressed in VPL.

2.3 Maintenance of Materialized Views

Many incremental view maintenance algorithms have been developed, for both centralized database systems [18, 10, 11] and distributed systems, such as data warehouses [2, 38, 32]. The main difference of view maintenance between a centralized and a distributed system is that in centralized environments, base and view data are in the same place; therefore, the former are always available for querying in order to maintain the latter consistently. On the other hand, in a data warehouse, the maintenance of the views may require the querying of the remote data sources, which may be unavailable for various reasons. Furthermore, the remote accesses may delay the process of maintenance.

The approach of [10] (and its generalization [11]) uses multiple active rules to incrementally maintain the materialized views or derived data, respectively. Our approach instead translates one deductive rule into a single active rule using a discrimination network matching technique [5, 6]. The main advantages of our approach are a) easier rule maintenance, b) centralised rule selection and execution control, c) straightforward implementation of traditional conflict resolution strategies of KBSs, and d) net effect of events. Furthermore, the performance comparison of the two approaches, in [7], shows that our approach is faster for incremental insertions and deletions as well as considerably faster for bulk inserts using set-oriented rule execution.

For the maintenance of the derived objects, we use a counting algorithm that is similar to the one described in [18]. However, they only use the counting algorithm for non-recursive views while for the recursive ones, they use a similar algorithm with [11]. We use the counting algorithm for recursive views as well since the calculation always terminates even for infinite derivation trees [7].

The approaches of [2, 38, 32] for view maintenance in data warehouses follow an approach similar to [10] although they might not use active rules explicitly. However, the functionality is the same. The work presented in [2, 38] is concerned with the maintenance anomalies that may arise when the maintenance algorithms query the base data at the data sources in order to maintain the views at the warehouse. The approach of [32] eliminates the need to query the data sources by replicating parts of the base data that are absolutely necessary for the maintenance of the views. The replicated base data along with the views are self-maintainable.

Our approach also creates self-maintainable views; therefore, there is no need to be concerned with the maintenance anomalies of [38]. Compared to [32], we do not put any special effort to infer which base data should be replicated because all the necessary information is "stored" inside the memories of the two-input events of the discrimination network (see Section 3.5.3). Furthermore, our approach handles also recursively defined views.

3. The InterBase^{KB} System

In this section we describe the architecture of the InterBase^{KB} system along with the functionality of each of its subsystems.

The InterBase^{KB} system extends the InterBase* multidatabase [28, 9] with a KB module (KBM) that is responsible for integrating the schema of the component database systems and for running the inference engine that materializes the views of the component databases inside the data warehouse. The overall system architecture is shown in Figure 1. The components of the InterBase^{KB} system are the following:

InterBase^{KB} Server. This server maintains data dictionaries and is responsible for processing InterSQL queries, as in the InterBase* system. Furthermore, it hosts the materialized views of the data warehouse.

InterBase^{KB} Clients. These are the old InterBase* clients that connect to the InterBase^{KB} server and issue InterSQL [29] queries against the component databases or the materialized views of the data warehouse which are stored inside the InterBase^{KB} server's database.

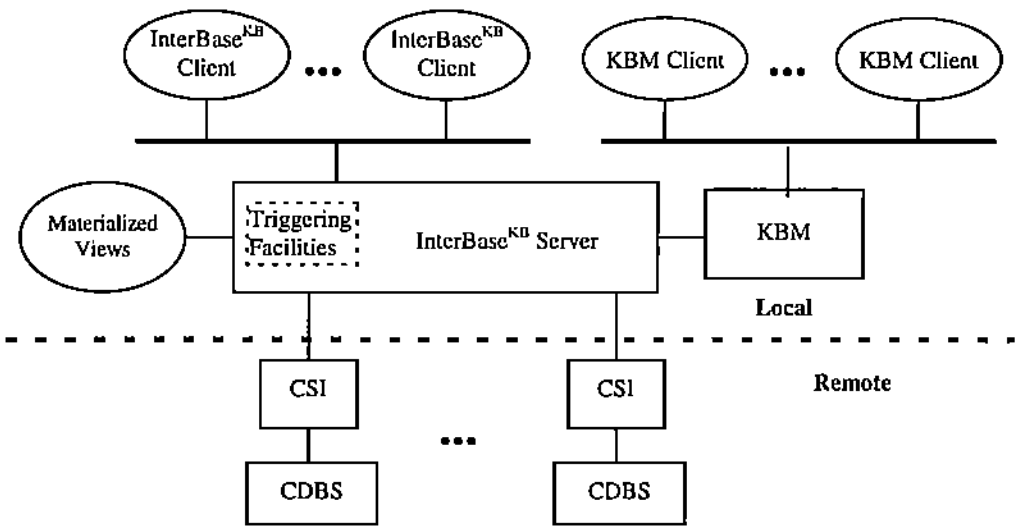

Figure 1. The Architecture of the InterBaseKB System

Knowledge Base Module (KBM). This module includes an active OODB, extended with declarative rules and an inference engine for a) integrating the schemes of the component databases and b) defining and maintaining the materialized views of the data warehouse.

KBM Clients. These clients have to be used in order to access the extended features of InterBase^{KB}, like global integrated schema, updateable materialized views, purely object-oriented database programming language, and declarative rules for programming expert database applications.

Component Database Systems (CDBSs). These are the heterogeneous systems that are integrated into the multidatabase. Furthermore, they are the data sources for the data warehouse.

Component System Interfaces (CSIs). These components act as an interface for the InterBase^{KB} server to the heterogeneous CDBSs.

3.1 The InterBase^{KB} Server

The InterBase^{KB} server hosts the materialized views of the data warehouse. That means that the users of the data warehouse need not access the base data of the CDBSs but can instead directly access the views provided for them inside the warehouse. However, the old application programs written for the nonfederated multidatabase system can still access the base data through the InterBase^{KB} server.

The server does not host the global integrated schema because this is defined and maintained inside the KB module, whose inferencing and data modeling capabilities are a superset of the capabilities of InterBase^{KB} server. However, if the administrator of the data warehouse chooses to materialize the integrated view of the CDBS base data, then these will be stored at the InterBase^{KB} server's database, and the integrated global schema will be hosted by the server.

The InterBase^{KB} server extends the InterBase* server with triggering capabilities. This means that when an InterBase^{KB} or a KBM client inserts, deletes or updates data in the InterBase^{KB} server's database, an event is raised that signals the occurrence of such a data modification action. This event is communicated to the KBM and possibly triggers an active or some declarative rule.

On the other hand, modifications to the data of the CDBSs are not captured by the triggering system of the InterBase^{KB} server but are handled by the CSIs as explained later, in Section 3.3. However, the changes that are detected at the CSIs level are propagated to the triggering subsystem of the InterBase^{KB} server, which is responsible for delegating it to the KBM for further processing

3.2 The InterBase^{KB} Client

The InterBase^{KB} clients are the clients of the nonfederated multidatabase system and kept to support the old applications. They connect to the InterBase^{KB} server and issue InterSQL queries against the component databases or the materialized views of the data warehouse which are stored inside the InterBase^{KB} server's database. They cannot be connected to the KBM because InterSQL cannot be translated to the fully object-oriented programming language of the KBM.

3.3 The Component System Interfaces

The CSIs act as an interface between the CDBSs and the InterBase^{KB} server. They translate InterSQL queries and commands to the native query language of the CSDB, and translate back the results, therefore they are version specific. While this is adequate for InterBase*, in InterBase^{KB} it is necessary for the interfaces to be able to detect changes of base data that have occurred inside the CDBSs by their native users and inform the InterBase^{KB} and the KBM subsequently that the data warehouse views might be inconsistent. It is the task of the KBM to decide and propagate these changes to the InterBase^{KB} server's database. However, it is the responsibility of the interface to detect the changes.

There are several ways to detect the data changes at the data sources, depending on the nature of the source itself. If the data source is a full-fledged database system, then the following solutions can be used:

- If the database system supports *triggering* or *active rule* facilities, these can be directly programmed through the CSI to directly notify data changes of interest.
- If the data source lacks active rule facilities, the next alternative is to inspect periodically the *log files* of the CDBSs to extract any interesting events.
- If the database system lacks both of the above features, the CSI can be programmed to periodically query the CSDB (polling) to detect any changes that have occurred since the last query. This can be very inefficient if the polling period is too low or very inaccurate if the polling is done infrequently and important changes are discovered too late.
- Finally, if the data (or information) source is not a database system but an application or a utility, periodic snapshots of the data can be provided and incrementally compared to detect the changes.

Regardless of the way the changes of data at the sources are detected, the communication of those changes to the data warehouse can either be done as soon as the change is detected or periodically. The latter solution can be configured to send the changes at the data warehouse when the latter is off-line, i.e. when it is not used for large decision support queries but runs in a maintenance mode. In this way, the maintenance of materialized data does not clutter the data warehouse during its normal operation.

3.4 The Knowledge Base Module

The *Knowledge Base Module* (KBM) is responsible for integrating the schema of the CDBSs and for running the inference engine that materializes the views of the component databases inside the data warehouse. The architecture of the KBM is shown in Figure 2. The components of the KBM are the following:

The Active Knowledge Base (A-KB) core. The KBM's core is an active object-oriented knowledge base system, called DEVICE [5, 6], which is built on top of the Prolog-based ADAM OODB [30, 17] and supports a) persistent objects, b) extensibility through metaclasses, and c) events and event-driven rules as first-class objects [13, 12]. More details on the A-KB are given later.

The A-KB is responsible for a) integrating the schemes of the component databases, b) defining and maintaining the materialized views of the data warehouse (stored at the InterBase^{KB} server), and c) propagating updates of the materialized views to the data sources.

The A-KB core communicates with the rest of the InterBase^{KB} system through a number of interface components. The administrator of the warehouse directly communicates with the A-KB core and can evoke methods for creating/destroying, enabling/disabling declarative rules for integrating the schemes of the component database systems and defining materialized views.

The OO-InterSQL interface. This interface translates the first-order rule definition language of A-KB into relational commands of InterSQL. Furthermore, it is responsible for translating simple object accessing methods into SQL retrieval/modification operations.

The Triggering Interface. This interface is responsible for capturing any data modification events trapped by either the triggering subsystem of the InterBase^{KB} server or the component system interfaces. The latter are not communicated directly to the KBM, but through the triggering subsystem of the InterBase^{KB} server. Therefore, the triggering interface of the KBM needs to capture only one event format. The events raised by the component system interfaces denote changes at the base data of the data sources while the events raised by InterBase^{KB} server denote changes made by the InterBase^{KB} or the KBM clients to the materialized views stored at the warehouse.

The KBM Client. This simple client accepts user queries interactively or user programs in batch mode and forwards them through the network to the KBM. The language used is Prolog extended with OO and persistence features, like OIDs, messages, etc.

The Storage System. The KBM needs to store data and methods, both for the user and for internal purposes, such as rule and event objects, the discrimination network memories, etc. The storage system is based on the built-in storage facilities of the underlying Prolog system, which is either ECLiPSe or SICStus Prolog.

3.5 The Active Knowledge Base Core

DEVICE integrates high-level, declarative rules (namely deductive and production rules) into an active OODB that

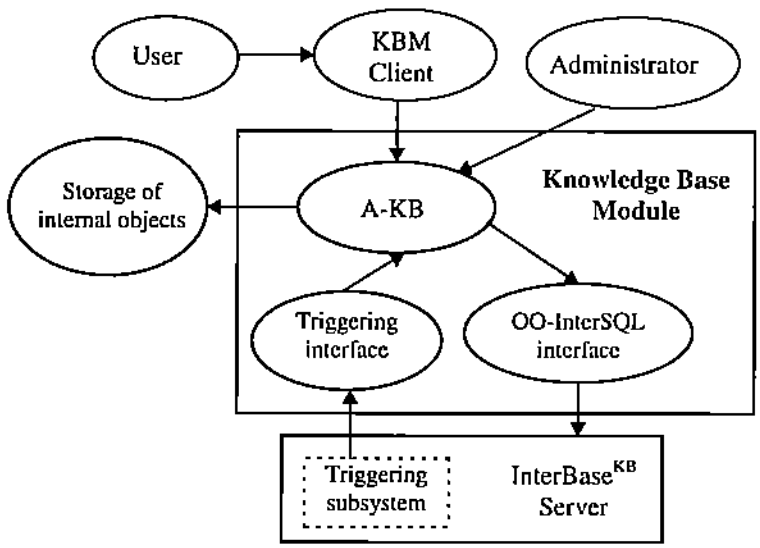

Figure 2. The Architecture of the Knowledge Base Module

supports only event-driven rules [12]. This is achieved by translating each high-level rule into one event-driven rule. The condition of the declarative rule compiles down to a complex event network that is used as a discrimination network that incrementally matches the rule conditions against the database.

In this section, we briefly describe the syntax and semantics for production rules which is the basis for integrating other types of declarative rules in the active OODB. Deductive rules and their use for view definition and maintenance are more thoroughly described in Section 4. Furthermore, we discuss the issues concerning rule compilation and condition matching. Further details about declarative rule integration in DEVICE can be found in previous work of ours [5, 6, 7].

3.5.1 Production Rule Syntax

Influenced by the OODB context that they are used, production rules mainly follow the OPS5 [15] paradigm. Rules are composed of *condition* and *action*, where the *condition* defines a pattern of objects to be detected over the database and *action* defines the set of updates to be performed on the database upon the detection of the pattern occurrence.

Example 1. The following rule deletes an employee named 'Mike' if his salary exceeds his/her manager's salary:

```
IF      E1@emp(name='Mike', salary:S1, manager:M1) and
        M1@emp(salary<S1)
THEN    delete ⇒ E1
```

The condition of a rule is an *inter-object* pattern that consists of the conjunction of one or more (either positive or negative) *intra-object* patterns. The intra-object pattern above denotes the conjunction of two instances of class emp that are connected to each other through two common attributes, denoted by the variables `M1`, `S1`.

The intra-object patterns consist of one or more *attribute* patterns. The first of the above intra-object patterns denotes an instance `E1` of class emp with attribute `name` equal to `'Mike'`, with salary `S1` and manager `M1`. The second intra-object pattern describes the manager `M1` of employee `E1`, who is himself/herself an instance of class emp, and his `salary` attribute is less than the salary `S1` of `E1`.

Variables in front of class names denote instances of the class. Inside the brackets, attribute patterns are denoted by relational comparisons, either directly with constants or indirectly through variables. Variables are also used to deliver values for comparison to other intra-object patterns (joins) in the same condition or to the action part of the rule. The variables are expressed as valid Prolog variables.

Path expressions inside attribute patterns are also allowed. The condition of the rule in Example 1 could be re-written as:

```
E1@emp(name='Mike', salary:S1, salary.manager<S1)
```

The innermost attribute should be an attribute of class emp, i.e. the class that the instances of the intra-object pattern stand for. Moving to the left, attributes should belong to classes related through object-reference attributes of the class of their predecessor attributes. We have adopted a right-to-left order of attributes, contrary to the C-like dot notation that is commonly assumed because we would like to stress the functional data model origins of [17]. Under this interpretation, the chained "dotted" attributes can be seen as function compositions.

During a pre-compilation phase, each rule that contains path expressions is transformed into one that contains only simple attribute expressions by introducing new intra-object patterns. The above pattern is actually transformed into the condition of the rule in Example 1.

Intra-object patterns can also denote classes instead of class instances. For example, the following intra-object pattern "checks" whether the class variable `task` of the class `production_rule` has the value `start`:

```
production_rule(task=start)
```

The difference between the two notations is just the presence of a variable in front of the class name. During parsing, the above pattern is rewritten as follows:

```
production_rule@production_rule_manager(task=start)
```

where `production_rule_manager` is the metaclass of `production_rule`. The new pattern is handled as a normal intra-object pattern.

There can also be negated intra-object patterns in the condition. A negated intra-object pattern denotes a negative condition that is satisfied when no objects in the database satisfy the corresponding positive intra-object pattern. For example, the following rule does not allow Mike alone to have the largest salary:

```
IF E1@emp(name='Mike', salary:S1) and
   not E2@emp(salary>S1) and
   prolog{NewSal is 0.9 * S1}
THEN   update_salary([S1,NewSal]) ⇒ E1
```

We notice that only safe rules [34] are allowed, i.e. a) variables that appear in the action must also appear at least once inside a non-negated condition and b) variables that are used inside a negated condition must also appear at least once inside a non-negated condition; otherwise they are just ignored.

The choice for the logic-like condition language is justified by the fact that the condition is supposed to be a declarative specification of the state of the database, and therefore, it is not appropriate to use the procedural interface of the OODB as the condition language. However, the use of arbitrary Prolog or ADAM goals to express some small static conditions or to compute certain values is allowed in the condition through the special `prolog{}` construct.

The action part of the rule is expressed in an extended Prolog language, enriched with the default procedural data manipulation language of ADAM. In the appendix, we include the full syntax of the condition-part language. The syntax of ADAM messages can be found in [17].

3.5.2 Production Rule Semantics

Rule conditions are matched against the database incrementally, using only the updated data. Rules are normally processed only at the end of a transaction or at user-defined *checkpoints*. However, the database updates are propagated through a discrimination network of complex events, as soon as the events happen, i.e. upon the successful completion of the updates. Multiple rules and/or rule instantiations can be eligible for execution (*firing*) at rule checkpoints. The set of all triggered rule instantiations is called the *conflict set*. Upon the detection of the checkpoint event (also issued by the system at the end of each transaction), the production rule manager checks the conflict set and selects only one rule instantiation for execution.

The selection is done according to the traditional OPS5 conflict resolution strategies, namely recency, specificity, and refractoriness. Other resolution strategies, such as priorities between rules and/or rule sets, can be easily implemented since the object-oriented nature of both the method and the implementation provides flexibility and extensibility.

When a rule is selected for firing, its actions are executed and the possible updates are propagated to the discrimination network in a manner similar to normal user updates. When all the actions are executed, the rule manager self-raises a checkpoint event to continue the production cycle until no more rule instantiations exist in the conflict set. After that, the control of the transaction is given back to the user.

3.5.3 Rule Compilation and Matching

According to [36], there exist ways to translate production rules into ECA rules because they are not really completely different paradigms but rather a different view of the same aspect. In this way, one can embed production rules into an active database system using the primitives of the latter.

Production rules are compiled to ECA rules, in order to be constantly monitored by the active database. The condition of a rule is compiled into a complex event network, which is associated with the event-part of the ECA rule, while the action-part of the ECA rule is the action of the production rule.

The condition compilation method uses complex events to translate a production rule into only one ECA rule. For example, the following (abstract) production rule: `IF a&b THEN action`, is translated into the following ECA rule: `ON` e_a`&`e_b `[IF true] THEN action`, where e_a, e_b are primitive events that detect the insertion of the data items `a`, `b`, respectively, and the operator `&` denotes event conjunction.

The complex event manager of the OODB monitors the above primitive events and combines their parameters in order to detect the occurrence of the complex event incrementally. The parameters of the currently signaled events are always combined with the parameters of the complete history of event occurrences, which are kept in event memories, in order not to miss any valid rule activation. When the complex event is detected, the condition of the rule has been matched, and the rule manager is responsible for scheduling it to execute.

Notice that the condition part of the ECA rule is always `true` because all conditions tests have been incorporated into the complex event. However, some small static conditions are allowed to be checked in the condition part of the ECA rule through the `prolog{}` construct.

The efficient matching of production rules is usually achieved through a discrimination network, such as RETE [16], TREAT [27], etc. DEVICE smoothly integrates a RETE-like discrimination network into an active OODB system as a set of first class objects by mapping each node of the network onto a complex event object of the active database system.

Each attribute pattern inside any intra-object pattern in the condition is mapped on a *primitive event* that monitors the insertion (or deletion) of values at the corresponding attribute. The insertion of data causes the *signaling* of an insertion event while the deletion of data causes the *anti-signaling* of a deletion event. In both cases, a token (positive or negative) with the parameters of the triggering event are propagated into the complex event network. Attribute comparisons with constants are mapped onto *logical events*, which perform simple attribute tests, and they are only raised when the associated condition is satisfied.

An intra-object pattern that consists of at least two attribute patterns is translated into a *two-input intra-object event* that joins the parameters of the two input events based on the OID the message recipient objects. Multiple intra-object events are joined in pairs based on the shared variables into *inter-object* events. The last inter-object event of the network maps the whole rule condition, and it is directly attached to the ECA rule that maps the original rule.

Two-input events receive tokens from both inputs whose behavior is symmetrical. The incoming tokens are stored at the input memories and are joined with the tokens of the opposite memory. According to a pre-compiled pattern, the join produces one or more output tokens which are propagated further to the event network. Stored tokens can be only explicitly deleted by the propagation of anti-signals in the network.

Finally, when the last event in the network signals, it means that the corresponding production rule condition is satisfied, and it must be fired. The rule instantiation token is then forwarded to the rule manager which stores it in the conflict set. The rest of the procedure has been described in the previous section. On the other hand, when an anti-signal is received by the rule manager, it means that a rule instantiation, if it still exists, must be deleted from the conflict set.

4. Deductive Rules for Object-Oriented View Definition and Maintenance

Schema integration in multidatabase and heterogeneous environments is usually achieved by defining common views of the underlying data. In this way, details of the heterogeneous data sources are abstracted away, and the user transparently sees a global schema. In this and the following sections, we thoroughly describe the view definition language of InterBase^{KB} along with the techniques for maintaining and updating the views.

In this section, we mainly focus on creating views in an OODB without taking into account the integration of heterogeneous data sources. In the next section, we describe how the basic view definition language is extended to cater for heterogeneous databases to be integrated.

The view definition language of InterBase^{KB} is provided by the deductive rules of the A-KB core, based mainly on Datalog [34]. Deductive rules describe data that should be in the database (intentional DB) provided that some other data and relationships among them hold in the current database state.

4.1 Deductive Rule Syntax

The syntax of deductive rules is very similar to the syntax of production rules (section 3.5.1), especially concerning the condition part which is identical. The action part of the production rule is replaced by the *derived class template* (DCT) which defines the objects that should be in the database when the condition-part is true.

Example 2. The following pair of deductive rules:

```
DR1:   IF A@arc(start:X,finish:Y)
       THEN path(start:X,finish:Y)
DR2:   IF P@path(start:X,finish:Y) and
          A@arc(start:Y,finish:Z\=X)
       THEN path(start:X,finish:Z)
```

defines a derived class `path` which includes objects with attributes `start`, `end` that have values that define the transitive closure of the `arc` relation.

Class `path` is a derived class, i.e. a class whose instances are derived from deductive rules. Only one DCT is allowed at the THEN part (head) of a deductive rule. However, there can exist many rules with the same derived class at the head. The final set of derived objects is a union of the objects derived by the two rules.

The DCT consists of attribute-value pairs where the value can either be a variable that appears in the condition or a constant. The syntax is given in the appendix.

4.2 Deductive Rule Semantics

Deductive rules are just an abstract way for defining new data in terms of existing or other derived data. The way the derivation is realized depends on the actual mechanism, the intended use of the derived data, and the frequency that the base data that they depend on are modified. In InterBase^{KB} we use deductive rules for defining and maintaining materialized views to be stored and re-used in a data warehouse independently from the data sources.

The semantics of deductive rules are thus similar to the semantics of production rules. When the condition of a deductive rule is satisfied, the object that is described by the derived class template is inserted in the database. When base data are modified, the rule's condition may not be any longer satisfied, therefore, changes to base data are propagated to the derived data. When a condition becomes false, the object that has been inserted in the database should be deleted. A counter mechanism is used to store the number of derivations for each derived object [18]. In this way, it can be checked whether the object to be deleted is still deducible by another rule instantiation.

Other possible semantics for deductive rules are goal driven rules which are activated when a query on the derived data is made. Deductive rules are then used to derive all the deducible objects; after the query is answered derived data do not persist. The set of derivable objects can be created using forward chaining techniques (like semi-naive evaluation, magic sets, etc.) or backward chaining, in the fashion of Prolog.

The A-KB core supports rules with such semantics, but in InterBase^{KB} we only use the materialized deductive rules which are most useful for data warehousing.

4.3 Type Derivation for Derived Classes

The type of the derived class, i.e. its structure and its placement in the class hierarchy, is determined by the deductive rules. More specifically, the DCT defines the structure of the derived class i.e. the number, names, and types of attributes and methods. Each new, derived object that is created fills these attributes with values that are derived by matching the rule condition. Furthermore, the derived classes contain some auxiliary attributes for assisting the derivation process, as described in the next sections.

The new derived class (or *d-class*) must be placed in the existing class hierarchy in order to seamlessly integrate the view in the existing database schema. There are two issues related with the placement of classes in the existing class hierarchy:

- How is the type of the new class (i.e. its signature of attributes and methods) related to the rest of the classes.
- How is the extent (set of instances) of the new class related to the extents of the rest of the classes.

In order to answer correctly to both the above questions, the separation of classes as extents and classes as types has been proposed [33]. In this way, two different hierarchies that do not interfere with each other can be created. However, this approach can only be used when views are not materialized but are just aliases for queries to the base data.

In our case, the views are materialized and the problem of placing the new class into the existing schema hierarchy is different. Specifically, the second of the above questions does not apply at all. If views are made subclasses of existing

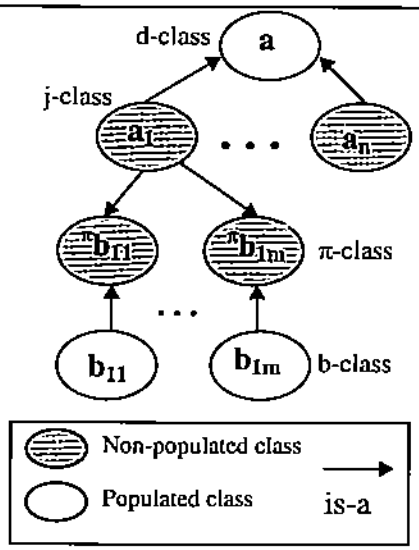

Figure 4. Class hierarchy for a derived class

classes, then the set of instances that belong to the superclass contains duplicates: a) the original instances of the base class (or *b-class*) and b) the instances of the view that are snapshots of the former. The only way to avoid that is to not relate the extents of the view with the extents of the b-classes.

The placement of the d-class in the class hierarchy depends on the conditions of the deductive rules. However, for a single d-class, there can be many deductive rules with different conditions. The scheme we propose in order to add the d-class into the class hierarchy is shown in Figure 4. To explain our type derivation scheme, consider the general case for defining a d-class through multiple deductive rules in Figure 3. Class a is defined through n deductive rules. Without loss of generality, we assume that the conditions contain only intra-object patterns b_{ij} of b-classes. Later we will raise this restriction.

Each intra-object pattern of class b_{ij} that exists in the condition is "mirrored" by a class $^{\pi}b_{ij}$ which actually represents the "projection" of class b_{ij} to the attributes of the intra-object pattern. This projection class (or *π-class*) exists for completeness even when no projection occurs and all the attributes of class b_{ij} correspond one-to-one to the attribute patterns of the intra-object pattern.

For example, the condition of rule DR_1 of Example 2 contains one intra-object pattern for arc class with or without projecting out some of the class attributes since the complete structure of this class may contain more attributes. Anyway the class $^{\pi}$arc exists in the class hierarchy created for this deductive rule.

The π-class is always a superclass of the b-class because it contains the same or less attributes than the b-class; therefore, its type is subsumed by the type of the b-class. There can exist multiple π-classes for a single b-class because there can exist many different intra-object patterns for the same b-class in multiple rules. When multiple π-classes exist, they can be organized in a hierarchy based again on type subsumption. When no subsumption between two π-classes exist, the two π-classes are both immediate superclasses of the b-class (multiple inheritance). This is allowed since no type conflicts exist for the common attributes of the two π-classes.

The condition of each deductive rule R_i may contain one or more conjuncted intra-object patterns b_{ij}. This conjunction is represented in the hierarchy graph through a class a_i that is a direct subclass of each one of the π-classes $^{\pi}b_{ij}$ of the intra-object patterns b_{ij}. The class a_i is called join-class (or *j-class*) because the conjunction of intra-object patterns is equivalent to joining the corresponding classes on the attributes denoted through the common variables. The structure of

R_1: IF $B_{11}@b_{11}(\{^iX_{11}\})$ and ... and $B_{1m}@b_{1m}(\{^iX_{1m}\})$
THEN $a(\{^iY \in \{^iX_{11}\} \cup \ldots \cup \{^iX_{1m}\})$
...
R_n: IF $B_{n1}@b_{n1}(\{^iX_{n1}\})$ and ... and $B_{n1}@b_{n1}(\{^iX_{n1}\})$
THEN $a(\{^iY \in \{^iX_{n1}\} \cup \ldots \cup \{^iX_{n1}\})$

Figure 3. The general case for defining a derived class

the j-class a_i contains the union of all the attributes of all the π-classes $^{\pi}b_{ij}$ of the condition; therefore, the set of attributes of each $^{\pi}b_{ij}$ is a subset of the attributes of a_i, which naturally makes a_i a subclass of all $^{\pi}b_{ij}$.

Finally, the d-class a is a direct superclass of all the j-classes a_i of the deductive rules R_i. The variables that appear in the DCT are a subset of the variables that appear in the rule condition (safety requirement); therefore, the type of each class a_1 subsumes the type of class a. For example, consider rule DR_2 of Example 2 where the condition contains three variables X, Y, Z, while the DCT projects away variable Z and contains only two variables X, Y.

Of course, it should be noted here that this scheme does not allow the renaming of the attributes from the b-classes to the d-classes. If, for example, rule DR_1 of Example 2 was as follows:

```
DR'1:   IF A@arc(start:X,finish:Y)
        THEN path(begin:X,end:Y)
```

then the d-class path could not possibly be a superclass of the j-class $path_1$ of the rule DR'_1, simply because the attribute names of the two classes are different. This problem is resolved by letting the j-classes inherit the actual "renamed" attributes of the d-class and then making the two versions of the same attribute "equal" through their retrieval methods.

In rule DR'_1, the j-class $path_1$ would have all four attributes (in order to keep the class hierarchy correct), and the retrieval method of e.g. start attribute would point to attribute begin.

One final remark is that the π-classes and the j-classes are just abstract classes, i.e. classes that do no have direct instances, but are mere structure and behavior placeholders. These classes are created to smoothly integrate the type of the d-class in the class hierarchy lattice of the OODB schema. Also notice that by using these classes, we avoid the problem of duplicate class instances through the subclass mechanism. The b-classes and the d-classes, which are directly instantiated, are not linked with a direct or indirect is-a link; therefore, the extents of the two classes are not connected through the subclass relationship.

The above property of not duplicating instances holds also for the abstract classes of the schema. For example, the instances of the π-classes include the instances of the b-classes and the instances of the j-classes. The latter, however, are abstract classes; therefore, π-classes have unique instances. The same holds for j-classes as well because they do not have any instances at all.

4.3.1 Recursive derived class definitions

So far we have only considered deductive rules that their condition contains only intra-object patterns of base classes. However, the most general case is that derived classes can appear as intra-object patterns in deductive rules as well. D-classes are never directly connected through an *is-a* link, even when a d-class appears inside the condition of a rule for another d-class. There is always a π-class and a j-class between them, as it can be seen in Figure 5.

The only "potential" problem with d-classes appearing in deductive rule conditions are the direct or indirect recursive derived class definitions which could introduce cycles in the class-hierarchy graph. For example, consider class path of Example 2 whose completed class hierarchy graph is shown in

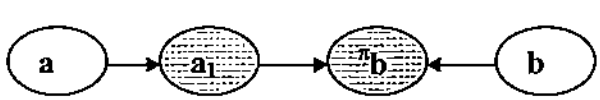

Figure 5. The simplest deductive rule

Figure 5.

It is evident that the direct recursion is denoted by a direct link from a j-class to a π-class, both "belonging" to the same d-class. Despite the recursion, the graph in Figure 6 is still acyclic which is a prerequisite for a correct class schema. We shall now prove that this property holds for any such class schema of derived classes which makes our type derivation method correct.

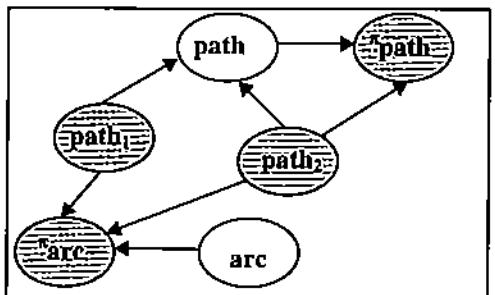

Figure 6. Class hierarchy for a recursive derived class definition

Theorem. The class hierarchy graph that includes π-classes, j-classes, and d-classes is acyclic.

Proof. By their definition, j-classes are direct subclasses of d-classes and π-classes, and they do not have subclasses themselves. Therefore, in the class-hierarchy graph, *is-a* arcs only flow in a j-class but do not flow out. Furthermore, *is-a* arcs emanated from j-classes are the only ones that flow in d-classes since d-classes cannot be directly linked with each other.

Now, let's assume that there is a cycle in the graph that contains at least one recursive d-class. All classes in the cycle must have at least one of each of flowing in and flowing out arcs. The recursive d-class is always connected with at least one j-class, as explained above. However, this j-class (that also belongs to the cycle) does not have arcs flowing in (explained above). Therefore, the cycle is broken.

4.4 View Materialization and Maintenance

It has been already mentioned that deductive rules are integrated using production rules. However, there are cases where the simple semantics of the latter are not adequate for capturing the semantics of the more high-level deductive rules and must be extended.

For example, the creation of a new derived object should only be done if the object does not already exist; otherwise, two distinct objects with the same attribute values will exist. This is a consequence of the generic differences between the OID-based OODBs and the value-based deductive databases [35].

Furthermore, when the condition of a previously verified deductive rule becomes false, the derived object of the head must be removed from the database. Before this is done, however, it must be ensured that the derived object is not deducible by another rule instantiation. For this reason, we use a counter mechanism which stores the number of derivations of an object [18]. If the derived object has a counter equal to 1, it is deleted; otherwise the counter is only decreased by 1.

The above operational semantics of deductive rules can be modeled by the following production rules:

```
IF condition and exists(object) THEN inc_counter(object)

IF condition and not(exists(object)) THEN create(object)

IF counter(object)>1 and not(condition) THEN dec_counter(object)

IF counter(object)=1 and not(condition) THEN delete(object)
```

In order to integrate the semantics of the above 4 rules into a single (extended) production rule, simple production rules are extend with an `anti_action` (or ELSE) part that hosts the derived object deletion algorithm. Using this extended scheme, a deductive rule can be modeled by a single production rule:

```
IF      condition
THEN    (if exists(object) then inc_counter(object) else create(object))
ELSE    (if counter(object)>1 then dec_counter(object) else delete(object))
```

Furthermore, the rule manager is extended to execute the anti-action upon the receipt of an anti-signal. Notice that the nested *if-then-else* constructs in the above rule are not production rules but the usual conditional primitive of the host programming language (Prolog).

The *conflict resolution* strategies of deductive rules also differ from production rules. The rule search space is navigated in a breadth-first or iterated strategy to model the set-oriented semi-naive evaluation of deductive rules [34], instead of the depth-first (recency) navigation of production rules. The execution order of rules with negation is determined by *stratification*, using the algorithm of [34].

The above deductive rule execution algorithms, combined with the incremental condition checking techniques and fixpoint semantics for production rules that have been described in Section 3.5, provide the mechanism for materializing and correctly maintaining the views over the data sources inside the data warehouse.

This incremental algorithm can be even used when the data source changes are communicated off-line in batches from the CSIs. The set of modifications are iterated and propagated throughout the network. The set of rule instantiations are executed under the fixpoint semantics, but only after all data source modifications have been considered.

An important advantage of using a discrimination network for the incremental maintenance of materialized views is that the views are self-maintainable [32]. This means that, in order to derive what changes need to be made to the materialized views in the warehouse when base data are modified, there is no need to query back the data sources [38]. All the necessary past data needed for maintaining the view is kept inside the memories of the two-input complex events.

4.5 Querying and Updating the Views

Derived classes allow both retrieval and modification operations on their instances. Retrieval can be done in two ways:

- Using normal OODB retrieval methods, like the ones used for any normal class [17].
- Using special methods that retrieve instances of derived classes according to their attribute values, like Prolog queries.

Furthermore, users and applications of the InterBase^{KB} clients can access the derived classes that are actually stored as relations in the InterBase^{KB} server, using InterSQL. However, we must notice here that the InterSQL queries and programs cannot access the full featured facilities of the OODB, such as method execution, composite objects, class inheritance, etc. In this way, the queries to materialized derived classes through InterSQL are restricted to single classes (not subclasses) and simple attributes (not OIDs).

The updates to the materialized views are usually forbidden since explicit creation, deletion, or modification of derived objects might violate the integrity of the views unless these modifications are propagated back to the source objects. Of course, the latter might be forbidden anyway in a data warehouse since the permission to update the source data from the warehouse depends on the ownership status of the data.

InterBase^{KB} provides a mechanism for updating the materialized views and propagating the updates to the base data, provided that such permission problems have been resolved and that the update of the source data from within the warehouse

is allowed. The mechanism of InterBase^{KB} is based on the preservation of the derivator-objects for a certain derived object. The derivator-objects for a derived object are the objects for which the condition of the deductive rule that derived a certain object is true.

Example 3. Consider the first rule of Example 2 and the following object of class `arc`:

```
17#arc@arc(start:node25,finish:node39)
```

where `17#arc` is the OID of the object. The above object makes the rule condition true and therefore an object:

```
3#path@path(start:node25,finish:node39)
```

for the d-class `path` is created. The object `17#arc` is the derivator for the derived object and is stored in the set-valued attribute `derivators` of `3#path` object:

```
derivators([[DR1,[3#path]]])
```

The OID of the rule object is stored along with the derivator to identify the rule that contributed to the derivation of the object. The tuple `[DR1, [3#path]]` is called *derivator tuple*.

Derivators are used for propagating the updates of derived objects back to the original base objects.

Example 4. If the following message:

```
update_start([node25,node26]) ⇒ 3#path
```

is sent to the derived object `3#path` then, based on the information that the derivator of this object is the object `17#arc`, the message can be forwarded to the derivator:

```
update_start([node25,node26]) ⇒ 17#arc
```

The latter will trigger the mechanism of maintaining the materialized view, described in the previous section, and the derived object `3#path` will be updated after the update of the base object. The re-direction of the original update message is done through the re-definition of the corresponding methods of the d-class.

The same method is followed almost exactly when there are more than one derivators for a single object.

Example 5. If the object `3#path` were derived from the second rule of Example 2 and the following two objects:

```
5#path@path(start:node25,finish:node35)
29#arc@arc(start:node35,finish:node39)
```

then the same update:

```
update_start([node25,node26]) ⇒ 3#path
```

can be safely propagated to the first object `5#path`, which is the derivator object that contributes the value of the attribute of the derived object that is being updated. This information is easily compiled from the deductive rule definition.

Things get more complicated when more than one derivators contribute for the same attribute of the derived object, i.e. the update is done on a shared-variable (or joined attribute).

Example 6. Consider the following rule:

```
IF    A1@arc(start:X,end:Y) and A2@arc(start:Y,end:Z)
THEN  path_length_two(start:X,medium:Y,end:Z)
```

If the update is done on the `medium` attribute that comes from the join on the shared variable `Y`, it is ambiguous whether the update will be propagated to the first, the second, or both derivators. All propagation actions are correct, and one of them must be chosen. Therefore, a tool that would identify those ambiguities to the data warehouse administrator/designer and ask which one of the alternatives is more appropriate would be very helpful. Currently in

InterBase^{KB}, the update is simply not allowed by re-defining the modification methods of the attribute to do nothing. However, the methods can be later manually re-defined to the desired alternative.

Finally, a similar ambiguity exists when there are multiple derivator tuples for a single object.

Example 7. If the object `3#path` is derived using both the derivators from Example 3 and Example 5:

`derivators{[[DR1,[3#path]],[DR2,[5#path,29#arc]]]}`

then any update message concerning any attribute of this `path` instance could be propagated to the objects of one or both derivation tuples. Again, the ambiguity can only be resolved manually or with a tool by the rule designer. Currently in InterBase^{KB}, there is a class attribute `propagation_rule` that stores the rule OID of the preferred deductive rule(s) to be used for the propagation. The re-defined modification methods of the d-class take into account the values of this slot before actually propagating the updates to the base objects.

5. Deductive Rules for Integrating Heterogeneous Data

In this section, we describe the mechanisms of InterBase^{KB} for integrating data from heterogeneous data sources. First, we briefly overview the requirements for such a mechanism. In the following sections, we show how these requirements are fulfilled from extensions to the deductive rule language and semantics.

The main requirements for integrating of heterogeneous data are the following.

Schema translation of the component databases. The various component databases or data sources probably have their own schemata which might have been expressed in different data models. Therefore, a mechanism is needed to translate the data model of each data source to the common data model of the data warehouse. InterBase^{KB} supports an object-oriented common data model [31] which is rich enough to capture the heterogeneity between the data models of the data sources.

Resolution of schematic and semantic conflicts. After the homogenization of the data models, there is still a need to resolve the conflicts among the schemata of the data sources. There can be many kinds of conflicts among the local schemata [31, 21, 8], such as schematic, semantic, identity, and data conflicts. The mechanism for schema integration should be general enough to be able to resolve most of them.

Integration transparency. After local schemata have been translated into the common data model and a single global schema exists, the users of the data warehouse should not know which data comes from which data source. Instead the system should distribute their requests transparently to the appropriate data source.

Throughout the section, we will use the following example of heterogeneous data sources.

Example 8.

Consider a federation of faculty databases in a university, consisting of OODBs `faculty_A`, `faculty_B` and `faculty_C`, corresponding to each of the three faculties `A`, `B` and `C`. Each database maintains information about the faculty's departments, staff, and the total number of employees per staff category. The schemata of the three databases are shown in Table 1.

Database	*faculty_A*	*faculty_B*
class	personnel	personnel
attributes	dept: deptID category: string no_of_emp: integer	dept: deptID $category_1$: integer $category_2$: integer ... $category_n$: integer

Database	*faculty_C*			
class	$category_1$	$category_2$	...	$category_n$
attributes	dept: deptID no_of_emp: integer	dept: deptID no_of_emp: integer		dept: deptID no_of_emp: integer

Table 1. Schemata of faculty databases

The faculty_A database has a single class personnel which has one instance for each department and each category of staff. The database faculty_B also has a single class personnel but staff category names appear as attribute names and the values corresponding to them are the number of employees per category. Finally, faculty_C has as many classes as there are categories and has instances corresponding to each department and the total number of employees for each category.

The heterogeneity of these databases is evident. The concept of staff categories is represented as atomic values in faculty_A, as attributes in faculty_B, and as classes in faculty_C. We assume that the names of the categories are the same in each database, without loss of generality, since it is not difficult to map different names using our deductive rule language.

5.1 Extensions to the Rule Syntax

In this section, we present the extensions introduced to the deductive rule language in order to cater for the integration of heterogeneous data.

5.1.1 External Schema References

```
DB_A:   IF     P@personnel/faculty_A(dept:D,category:C,no_of_emp:N)
        THEN   personnel(dept:D,category:C,no_of_emp:N)

DB_B:   IF     P@personnel/faculty_B(dept:D,C\=dept:N)
        THEN   personnel(dept:D,category:C,no_of_emp:N)

DB_C:   IF     P@C/faculty_C(dept:D, no_of_emp:N)
        THEN   personnel(dept:D,category:C,no_of_emp:N)
```

Figure 7. Deductive rules for integrating the schemata of Example 8

An external relational or OODB schema is translated into InterBase^{KB} as a collection of classes. The schema of the class is the same as the schema of the corresponding external relation or class, concerning the names and types of attributes. A relation/class is imported in InterBase^{KB} using a deductive rule for defining a derived class as a "mirror" of the external entity. The external (base) class is represented in the condition of the rule using the normal rule syntax extended with a reference to the name of the external database.

The class `personnel` of database `faculty_A` of Example 8 is imported into InterBase^{KB} as shown in Figure 7. The name of the database from which the relation/class is imported appears just after the name of the class. The interpretation of this reference to an external database will be presented in section 5.2.1.

5.1.2 Second-order Syntax

The derived class `personnel` will be also used to import personnel data from the rest of the faculty databases. However, the import of the other databases cannot be done in such a straight forward manner because the staff categories are either attribute or class names, and a second order syntax is needed. When variables of a deductive rule language can range over attribute or class names, we say that the rule language has a second-order syntax. The databases for `faculty_B` and `faculty_C` of Example 8 are imported as shown in Figure 7.

Rule DB_B has a variable `C` that ranges over all the attributes of the class `personnel` of database `faculty_B`, except attribute `dept`, which is explicitly mentioned in the condition. Rule DB_C has again a variable `C` that ranges over the classes of database `faculty_C`. Despite the second-order syntax, the above rules are interpreted using a set of first-order rules, as it will be described in section 5.2.2.

5.2 Schema Integration

In this section, we describe how the deductive rule language extensions are integrated in the view definition and maintenance mechanism of InterBase^{KB} providing schema integration for heterogeneous data sources.

5.2.1 Importing External Schemata

Each imported database is represented in InterBase^{KB} as a metaclass. This metaclass contains all the necessary information about the imported database, such as its name, type, network address of CSI and CDB, exported relation/classes, communication and/or storage protocols, etc. This information is copied from the system's data directory [28].

Each relation/class that is imported from an external database is an instance of the above metaclass. In this way, the information about the origins of a specific class can be easily traced by following the `is_instance_of` link. Figure 8 shows how the databases and classes of Example 8 have been imported in InterBase^{KB}. It is obvious that the name of the imported database metaclass is constructed by appending the `meta_class` keyword to the name of the database while the names of the imported classes are constructed by concatenating the original class/relation name and the name of the database. Of course, the renaming is done automatically by the system.

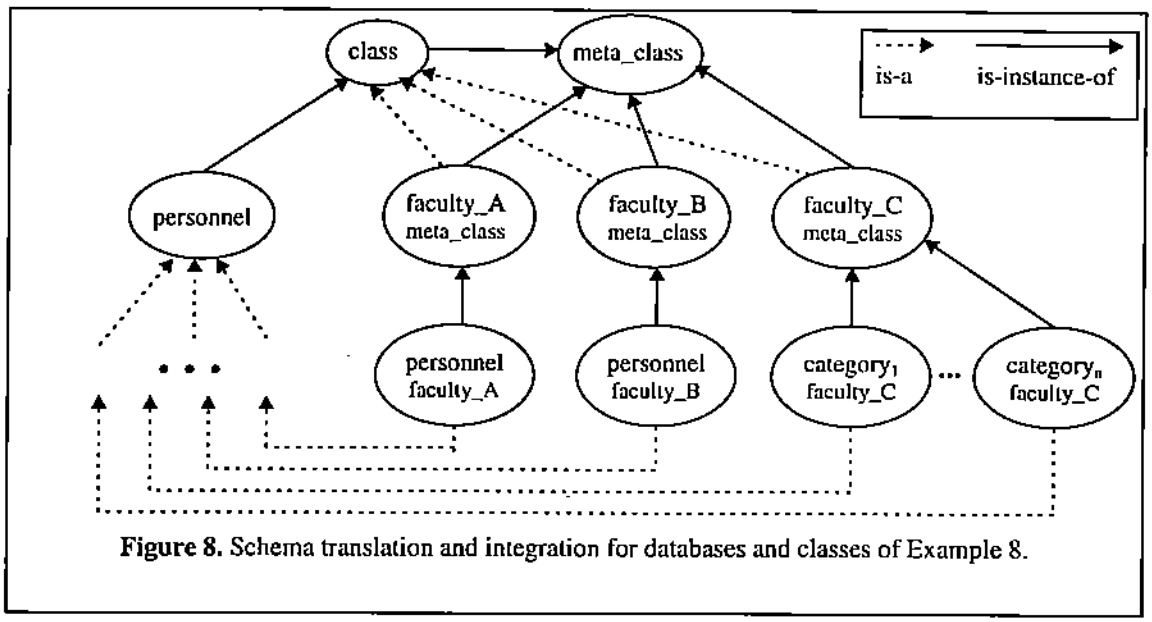

Figure 8. Schema translation and integration for databases and classes of Example 8.

The imported classes are now considered base classes when defining the common view for personnel data. Therefore, they are connected to the d-class `personnel` with intermediate classes that serve type preservation purposes, as explained in Section 4.3. In Figure 8, the details about these intermediate classes are skipped.

5.2.2 Semantics of the Extended Rule Syntax

In this section, the translation of the extended rule syntax will be described. The most important and difficult is the translation of the second order syntax into equivalent rules with first-order semantics. This is achieved through the metaclasses of the OODB schema. Each second-order reference for OODB classes in a rule is transformed into a first-order reference to the equivalent OODB metaclasses. Furthermore, a deductive rule that contains a second-order syntax is transformed into a production rule that creates first-order deductive rules. Figure 9 shows the translated deductive rules of Figure 7.

The translation of the first rule (DB'_A) is straight forward because it does not contain second-order constructs but only a reference to an external database. As it was explained in the previous section, the classes of the external databases are automatically renamed, thus appending the name of the database.

The other two rules require a translation of their second-order syntax. Concerning the condition, the intra-object (class) pattern is replaced with a pattern of the metaclass of the class. The new metaclass pattern can match its instances which are the classes to be "discovered" by the second-order constructs of the original rule. The attribute patterns of the original rule are transformed in attribute tests of the `slot_desc` attribute of the metaclass. This set-valued attribute is present in every metaclass [17] and contains the description (name, type, cardinality, visibility, etc.) for each attribute of its instances (classes).

The condition of the second rule that contains a variable for an attribute name ($\texttt{category}_i$) is directly translated into a metaclass pattern whose `slot_desc` attribute is retrieved and propagated to the rule action. Thus, the variable `C` stands now for a value of the `slot_desc` attribute, and the second-order construct is transformed into first-order. Since the

```
DB'A:   IF      P@personnel_faculty_A(dept:D,category:C,no_of_emp:N)
        THEN    personnel(dept:D,category:C,no_of_emp:N)

DB'B:   IF      personnel_faculty_B@faculty_B_meta_class(slot_desc:C\=dept)
        THEN    new_rule('IF P@personnel_faculty_B(dept:D,C:N)
                          THEN personnel(dept:D,category:C,no_of_emp:N)')
                ⇒ deductive_rule

DB'C:   IF      C@faculty_C_meta_class(slot_desc⊇[dept,no_of_emp]) and
                prolog(string_concat(C,'_faculty_C',C1))
        THEN    new_rule('IF P@C1(dept:D, no_of_emp:N)
                          THEN personnel(dept:D,category:C,no_of_emp:N)')
                ⇒ deductive_rule
```

Figure 9. Translation of deductive rules of Figure 7

class name (`personnel_faculty_B`) is known from the original rule, the instance of the metaclass (`faculty_B_meta_class`) in the metaclass pattern is instantiated.

The condition of the third rule is slightly different since the class name is a variable in the original rule. Therefore, the variable now appears as the OID of the instance of the metaclass pattern. Furthermore, the selection of the classes among the instances of the metaclass (`faculty_C_meta_class`) is restricted to those that have at least the attributes `dept` and `no_of_emp`.

The transformed rules are production rules (Section 3.5) and their action is a method call to create a new deductive rule. The deductive rule has a first-order syntax since any variables that appear in the place of attributes or classes have already been instantiated by the condition with the metaclass pattern. Rule DB'_C also contains a call to a Prolog built-in predicate in order to construct the proper name for the $category_i$ classes. Similar calls are also included in the actual implementation for creating the rule strings (e.g. for incorporating the variables), but are omitted here to ease the presentation.

The production rules of Figure 9 are triggered even if they are generated after the creation of the class and metaclass schema because the A-KB core includes a rule activation phase at the end of rule creation. Furthermore, the A-KB core creates events for every method of the OODB schema, including metaclasses. Rules DB'_B, DB'_C will be fired as many times as the number of categories in the respective databases and the same number of deductive rules will be generated. The new deductive rules will also be activated and fired based on the same mechanism. Rule DB'_A is a deductive rule and it will behave as described in Section 4.

5.2.3 View Management

The view that has been created using the set of deductive rules is managed exactly the same as described in Section 4. The user is unaware of the local schemata, and he/she is supposed to query only the top-level class of the view. Through this derived class, all imported databases have a common schema. This is called integration transparency. This common view can be used by other rules in their condition, and it will be considered as a base class.

The materialization, querying, updating, and maintenance of the view is done in the same way as for the rest of the views. Of course, the materialization of the base data means that all the data sources are mirrored inside the data warehouse, which wastes a lot of space. A typical solution [32] for this is to define a common view that is actually used in the condition of another deductive rule, i.e. to project away all the unneeded attributes of the local schemata. This will reduce the space needed for materializing the base data.

A more drastic solution we have adopted for InterBase^{KB} is not to materialize any attribute of the base data but rather to create derived objects which 'host' only their derivators (see Section 4.5). The information about the derivator objects is used for two purposes:

- to propagate updates from the view to the base data (stored at the local databases) as described in Section 4.5.
- to query the base data in the local databases when the user of the warehouse wishes so. This is done in a similar way with updating views. When a retrieval method (query) is received by an object of the view, then the query is forwarded (through re-definition of the basic methods) to the derivator objects in order to retrieve the attributes of the base object or tuple.

The maintenance of such materialized views requires only the generation and/or deletion of the derived objects and the insertion/deletion of derivator tuples in the `derivators` attribute. When base data are updated, the modifications are propagated to the warehouse but not actually stored in the attributes of the warehouse base data. However, the relevant events that trigger the deductive rules are raised, and the parameters of the modifications are propagated in the discrimination network.

One problem with the implementation of this technique is that the base data belong to different databases of possibly different data types. However, these base data must have a unique OID inside the data warehouse in order to be handled through the object-oriented model of the A-KB core. The "generation" of unique OIDs for all the imported objects is done through functions of the identifiers of the real objects [31] (in case of local OODBs) or through unique mappings of the keys of relational tuples [21] (in case of local relational DBs). These OID generating functions are actually methods stored at the metaclass of each imported database (section 5.2.1) and are evoked through wrappers that change the default OID generation and retrieval functions of the A-KB core.

6. Conclusions and Future Work

Data Warehouses are information repositories that integrate data from possibly heterogeneous data sources and make them available for decision support querying and analysis using materialized views. The latter need to be maintained incrementally in order to be consistent with the changes to the data soucres. Multidatabase systems are confederations of pre-existing, autonomous, and possibly heterogeneous database systems.

In this paper, we have presented the extension of a nonfederated multidatabase system with a knowledge-base module (KBM) providing a system with the necessary functionality to be used for data warehousing. The multidatabase system integrates various heterogeneous component databases with a common query and transaction specification language, but does not provide schema integration.

The KBM provides a declarative logic language which a) offers schema integration for heterogeneous data sources and b) allows the definition of complex views in the warehouse over the base data of the data sources. At the core of the KBM lies an active OODB that:

- supports metaclass hierarchies which allow the customization of schema translation and integration,
- supports events and event-driven rules,
- integrates declarative (deductive and production) rules, which allow the materialization and self-maintenance of the complex views, and
- provides second-order rule language extensions, which allow the declarative specification for integrating the schemata of heterogeneous data sources into the warehouse.

The views are self-maintainable in a sense that the data sources are not queried by the incremental view-maintenance mechanism which uses only the changes to the data sources and the information stored at the discrimination network that selects matching deductive rules. Furthermore, the views are updateable using a mechanism that stores the identifiers of the derivators of the derived objects. The latter mechanism is also used to not import in the data warehouse all the base data of the data sources.

All the above features, combined with the interoperability offered at the application level and the high-level support for an integrated atomic commitment by the multidatabase system, make a powerful yet flexible environment for data warehousing. Of course, it must be understood that the described system offers only the infrastructure for data warehousing and should not be considered as a full warehouse system. The latter requires several back-end and front-end tools to ease the task of designing, administering, maintaining, and querying the data warehouse.

We are currently investigating the support for OLAP and data cubes in the warehouse. The latter requires advanced data modeling capabilities, efficient storage and implementation mechanisms, and a powerful query language. In [7] we describe the extension of the logic-based language of InterBase^{KB} with aggregate attributes, using an efficient event-driven mechanism to maintain those attributes. We believe that such an extended view definition language along with the powerful object-oriented data model of the KBS provides just enough power and usability for a data warehousing environment.

Appendix - Declarative Rule Syntax

```
<production_rule> ::= if <condition> then <action>
<deductive_rule> ::= if <condition> then <derived_class_template>
<condition> ::= <inter-object-pattern>
<inter-object-pattern> ::= <condition-element> ['and' <inter-object-pattern>]
<inter-object-pattern> ::= <inter-object-pattern> 'and' <prolog_cond>
<condition-element> ::= ['not'] <intra-object-pattern>
<intra-object-pattern> ::= [{<variable>|<class>}'@']<class>['('<attr-patterns>')']
<attr-patterns> ::= <attr-pattern>[','<attr-patterns>]
<attr-pattern> ::= <attr-function> {':'<variable> | <predicates> |
                                   ':'<variable> <predicates> }
<predicates> ::= <rel-operator> <value> [{ & | ; } <predicates>]
<predicates> ::= <set-operator> <set>
```

<rel-operator> ::= = | > | >= | =< | < | \=

<set-operator> ::= ⊂ | ⊆ | ⊃ | ⊇ | ⊄ | ∈ | ∉

<value> ::= <constant> | <variable>

<set> ::= '[' <constant>[,<constant>] ']'

<attr-function> ::= {[<attr-function>'.']<attribute> | <variable> }

<prolog_cond> ::= 'prolog' '('<prolog_goal>')'

<action> ::= <prolog_goal>

<derived_class_template> ::= <derived_class>'('<templ-patterns>')'

<templ-patterns> ::= <templ-pattern> [',' <templ-pattern>]

<templ-pattern> ::= {<attribute>|<variable>}':'<value>

<class> ::= *An existing class or meta-class of the OODB schema*

<derived_class> ::= *An existing derived class or a non-existing base class of the OODB schema*

<attribute> ::= *An existing attribute of the corresponding OODB class*

<prolog_goal> ::= *An arbitrary Prolog/ADAM goal*

<constant> ::= *A valid constant of an OODB simple attribute type*

<variable> ::= *A valid Prolog variable*

References

[1] S. Abiteboul and A. Bonner, "Objects and Views," *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, 1991, pp. 238-247.

[2] D. Agrawal, A.E. Abbadi, A. Singh, and T. Yurek, "Efficient View Maintenance at Data Warehouses," *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, Tucson, Arizona, 1997, pp. 417-427.

[3] R. Ahmed, P. DeSmedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, and M.C. Shan, "The Pegasus heterogeneous multidatabase system," *IEEE Computer*, Vol. 24, No. 12, 1991, pp. 19-27.

[4] T. Barsalou and D. Gangopadhyay, "M(DM): An open framework for interoperation of multimodel multidatabase systems," *Proc. IEEE Int. Conf. on Data Engineering*, 1992, pp. 218-227.

[5] N. Bassiliades and I. Vlahavas, "DEVICE: Compiling production rules into event-driven rules using complex events," *Information and Software Technology*, Vol. 39, No. 5, 1997, pp. 331-342.

[6] N. Bassiliades and I. Vlahavas, "Processing production rules in DEVICE, an active knowledge base system," *to appear in Data & Knowledge Engineering*, 1997.

[7] N. Bassiliades, I. Vlahavas, and A. Elmagarmid, *E-DEVICE: An Extensible Knowledge Base System with Multiple Rule Support*, tech. report, Dept. of Computer Science, Purdue University, W. Lafayette, Indiana, September, 1997.

[8] C. Batini, M. Lenzerini, and S.B. Navathe, "Comparison of methodologies for database schema integration," *ACM Computing Surveys*, Vol. 18, No. 4, 1986, pp. 323-364.

[9] O. Bukhres, J. Chen, W. Du, and A. Elmagarmid, "InterBase: An Execution Environment for Heterogeneous Software Systems," *IEEE Computer*, Vol. 26, No. 8, 1993, pp. 57-69.

[10] S. Ceri and J. Widom, "Deriving production rules for incremental view maintenance," *Proc. Int. Conf. on Very Large Databases*, Morgan Kaufman, Barcelona, Spain, 1991, pp. 577-589.

[11] S. Ceri and J. Widom, "Deriving incremental production rules for deductive data," *Information Systems*, Vol. 19, No. 6, 1994, pp. 467-490.

[12] O. Diaz and A. Jaime, *EXACT: An extensible approach to active object-oriented databases*, tech. report, Dept. of Languages and Information Systems, University of the Basque Country, San Sebastian, Spain, 1994.

[13] O. Diaz, N. Paton, and P.M.D. Gray, "Rule management in object oriented databases: A uniform approach," *Proc. Int. Conf. on Very Large Databases*, Morgan-Kaufman, Barcelona, Spain, 1991, pp. 317-326.

[14] A.K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz, "A Multidatabase Transaction Model for InterBase," *Proc. Int. Conf. on Very Large Databases*, Brisbane, Australia, 1990, pp. 507-518.

[15] C.L. Forgy, *OPS5 User Manual*, tech. report, Dept. of Computer Science, Carnegie-Mellon University, 1981.

[16] C.L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, Vol. 19, 1982, pp. 17-37.

[17] P.M.D. Gray, K.G. Kulkarni, and N.W. Paton, *Object-Oriented Databases, A Semantic Data Model Approach*, Prentice Hall, London, 1992.

[18] A. Gupta, I.S. Mumick, and V.S. Subrahmanian, "Maintaining views incrementally," *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, 1993, pp. 157-166.

[19] S. Heiler and S. Zdonik, "Object Views: Extending the vision," *Proc. IEEE Int. Conf. on Data Engineering*, 1990, pp. 86-93.

[20] M. Kaul, K. Drosten, and E.J. Neuhold, "ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views," *Proc. IEEE Int. Conf. on Data Engineering*, 1990, pp. 2-10.

[21] W. Kim, I. Choi, S. Gala, and M. Scheevel, "On resolving schematic heterogeneity in multidatabase systems," *Distributed and Parallel Databases*, Vol. 1, No. 3, 1993, pp. 251-279.

[22] W. Kim and W. Kelley, "On View Support in Object-Oriented Databases Systems," *Modern Database Systems: The Object Model, Interoperability, and Beyond.*, W. Kim, Ed. ACM Press and Addison-Wesley, 1995, pp. 108-129.

[23] R. Krishnamurthy, W. Litwin, and W. Kent, "Language features for interoperability of databases with schematic discrepancies," *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, 1991, pp. 40-49.

[24] e. Kuehn, F. Puntigam, and A.K. Elmagarmid, "Multidatabase Transaction and Query Processing in Logic," *Database Transaction Models for Advanced Applications*, A. K. Elmagarmid, Ed. Morgan Kaufmann, 1991, pp. 298-348.

[25] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian, "On the logical foundation of schema integration and evolution in heterogeneous database systems," *Proc. Int. Conf. on Deductive and Object-Oriented Databases*, Springer-Verlag, Phoenix, Arizona, 1993, pp. 81-100.

[26] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian, "SchemaSQL - A Language for Interoperability in Relational Multi-database Systems," *Proc. Int. Conf. on Very Large Databases*, Morgan Kaufmann, Mumbai, India, 1996, pp. 239-250.

[27] D.P. Miranker, "TREAT: A better match algorithm for AI production systems," *Proc. AAAI*, 1987, pp. 42-47.

[28] J. Mullen, O. Bukhres, and A. Elmagarmid, "InterBase*: A Multidatabase System," *Object-Oriented Multidatabase Systems*, O. Bukhres and A. K. Elmagarmid, Eds., Prentice Hall, 1995, pp. 652-683.

[29] J.G. Mullen and A. Elmagarmid, "InterSQL: A Multidatabase Transaction Programming Language," *Proc. Workshop on Database Programming Languages*, 1993, pp. 399-416.

[30] N.W. Paton, "ADAM: An object-oriented database system implemented in Prolog," *Proc. British National Conf. on Databases*, CUP, 1989, pp. 147-161.

[31] E. Pitoura, O. Bukhres, and A. Elmagarmid, "Object Orientation in Multidatabase Systems," *ACM Computing Surveys*, Vol. 27, No. 2, 1995, pp. 141-195.

[32] D. Quass, A. Gupta, I.S. Mumick, and J. Widom, "Making Views Self-Maintainable for Data Warehousing," *Proc. Conf. on Parallel and Distributed Information Systems*, Miami, Florida, USA, 1996, pp. .

[33] M.H. Scholl, C. Laasch, and M. Tresch, "Updatable Views in Object-Oriented Databases," *Proc. Int. Conf. on Deductive and Object-Oriented Databases*, Springer-Verlag, Munich, Germany, 1991, pp. 189-207.

[34] J. Ullman, *Principles of Database and Knowledge-Base Systems*, Computer Science Press, Rockville, Maryland, 1989.

[35] J. Ullman, "A comparison between deductive and object-oriented database systems," *Proc. Int. Conf. on Deductive and Object-Oriented Databases*, Springer Verlag, Munich, 1991, pp. 263-277.

[36] J. Widom, "Deductive and active databases: Two paradigms or ends of a spectrum?," *Proc. Int. Workshop on Rules in Database Systems*, Springer-Verlag, Edinburgh, Scotland, 1993, pp. 306-315.

[37] J. Widom, "Special Issue on Materialized Views and Data Warehousing," in *IEEE Data Engineering Bulletin*, vol. 18(2), D. Lomet, Ed., 1995.

[38] Y. Zhuge, H. Garcia-Mollina, J. Hammer, and J. Widom, "View maintenance in a warehousing environment," *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, San Jose, California, 1995, pp. 316-327.